SecureJS Compiler: Portable Memory Isolation in JavaScript*

Yoonseok Ko
Inria
Sophia-Antipolis, France
yoon-seok.ko@inria.fr

Tamara Rezk
Inria
Sophia-Antipolis, France
tamara.rezk@inria.fr

Manuel Serrano
Inria
Sophia-Antipolis, France
manuel.serrano@inria.fr

ABSTRACT

The memory isolation mechanism plays an essential role to provide security enforcement in JavaScript programs. Existing secure interaction libraries such as Google Caja, SES, and VM2 rely on built-in memory isolation mechanisms provided by Node.js and browsers, yet most of the other engines such as JerryScript and Duktape, which are implementations for IoT devices, do not support such isolation mechanisms.

In this paper, we report about the design and implementation of SecureJS, a portable JavaScript-to-JavaScript compiler that enforces memory isolation. As it only uses standard features, the compiled code it generates can be used by any JavaScript engine. We validated empirically the semantics preservation and memory isolation of SecureJS compiled programs by using 10,490 test programs of ECMAScript Test262 test suite. We also developed a novel experiment to evaluate memory isolation property of compiled code by instrumented JavaScript engines.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Web application security**; • **Software and its engineering → Syntax**; *Semantics*;

KEYWORDS

Compiler, Memory Isolation, JavaScript

ACM Reference Format:
Yoonseok Ko, Tamara Rezk, and Manuel Serrano. 2021. SecureJS Compiler: Portable Memory Isolation in JavaScript. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3412841.3442001

1 INTRODUCTION

JavaScript has been growing at a rapid pace over the years due to its easy manipulation and deployment in development. The dynamic nature of the language provides a smooth environment for the integration of code, and it contributes to the easy and fast development of complex applications [10].

Whereas the dynamic features of JavaScript play an essential role in the language popularity by simplifying application developments, they also enable malicious JavaScript libraries to exploit these dynamic features to break the integrity and confidentiality of the applications [3]. Within a single JavaScript program execution all components share a global object and intrinsic objects such as `Object` and `Object.prototype`. Due to the idiosyncratic JavaScript semantics whose primitive operations implicitly access these global and intrinsic objects, all the components that form an application are tightly bond together. This gives opportunities to a malicious third-party code to pervert or extract undue pieces of information from others execution.

Various researches have been carried out so as to provide security enforcement achieved by memory isolation [2, 8, 9, 11, 13–16]. Total isolation is the simplest way to run programs without unintended interplay. However, more complex programs that need interactions between programs cannot enjoy the benefit of isolation. The practical secure JavaScript programming solutions such as Google Caja [9], SES [1], and VM2 [13] rely on such an isolation mechanism provided by Node.js or browsers and provide secure interactions between isolated programs by controlling the interactions. Still, such built-in isolation mechanisms are not supported other popular JavaScript implementations such as JerryScript[1] and Duktape[2], which are JavaScript implementations for micro-controllers, and thus, it limits the use of these secure interaction solutions in a practical use. It motivates the need for portable isolated JavaScript environments independent from JavaScript implementations.

A basic building block for JavaScript memory isolation is a *realm* (as it is defined in the latest ECMAScript specification[3]), which consists of a global object and intrinsic objects. Programs running in the same realm share the global and intrinsic objects, and isolated realms initially do not share any references between the realms. Running programs in different isolated realms guarantees that their memory accesses do not interfere with each other. However, programs in a realm are able to interact with programs in another realm by intentionally exporting a value, and it breaks the isolation. Thus, isolated realms are a basic and essential unit for secure program executions, and they can achieve security enforcement on further interactions by combining with the existing secure JavaScript programming solutions. Unfortunately, JavaScript realms are only supported by some JavaScript implementations, such as Node.js and browsers, and it limits the use of the existing secure programming solutions in other JavaScript implementations.

*This research has been partially supported by the ANR17-CE25-0014-01 CISC project, the Inria Project Lab SPAI, and the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892.

SAC'21, March 22-March 26, 2021, Gwangju, South Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442001

[1] https://jerryscript.net
[2] https://duktape.org
[3] https://www.ecma-international.org/ecma-262/#sec-code-realms

Our Approach. In this paper, we propose a novel technique, based on a JavaScript-to-JavaScript compilation, to run JavaScript programs in an isolated realm. Our method focuses on rewriting the semantics of implicit field accesses of the given program and achieves memory isolation by using revised field access operations. For that, we define a subset of JavaScript language, $JS^{explicit}$, in which primitive operations do not involve implicit field accesses. The translation from JavaScript to $JS^{explicit}$ replaces all the implicit field accesses of the source program with explicit operations. $JS^{explicit}$ programs are meant to run in a dedicated secure runtime environment that creates an isolated virtual realm and provides secure primitive operations that implicitly access the isolated virtual realm.

We have implemented a secure JavaScript-to-JavaScript compiler, SecureJS. It transforms programs into equivalent ones, but which would execute as if they were running inside an isolated memory. The compilation relies on the translation to $JS^{explicit}$ and replaces primitive operations to secure versions and thus execute in an isolated memory, which simulates a JavaScript realm. The program running in the isolated memory can interact with others by intentionally exporting a value of the isolated memory to outside. This compiler is portable and only uses standard features so it can be used by any JavaScript engine that complies with ECMAScript 5.1 specification. Thus, the compiler provides an isolation mechanism independently of JavaScript implementation, and it enables us to use existing secure interaction libraries in any JavaScript engine.

Contributions.

- We propose a subset of the JavaScript language $JS^{explicit}$, in which primitive operations do not involve implicit field accesses, and we present a translation from JavaScript to $JS^{explicit}$ (Section 3).
- We provide a secure JavaScript-to-JavaScript compiler, SecureJS, as a tool. This compiler transforms programs into equivalent ones but which would execute as if they are running inside an isolated realm without relying on any built-in mechanisms. This implementation is portable to any ECMAScript 5.1 compliant implementations and the isolated realm supports all features of ECMAScript 5.1[4] (Section 4).
- We provide instrumented JavaScript engines that detect implicit field accesses and field accesses to intrinsic objects. We have used it to test empirically the semantic preservation and memory isolation of the SecureJS compiled programs. The instrumented engines can be used independently of SecureJS to test memory isolation. Our tool and the patch for the instrumented engine are publicly available[5] (Section 5).

2 OVERVIEW

We give an overview of our approach with an overly simple example that only use a field access operation that involves a dynamic type conversion and prototype chain field lookup.

We demonstrate the evaluation of a JavaScript program construct. An evaluation of a single JavaScript program construct consists of a series of JavaScript *internal primitive operations*. We consider the following code fragment:

```
1 Boolean.prototype.x = 10;
2 var o = {toString: function() { return "x"; }};
3
4 true[o] === 10;
```

The evaluation of the field lookup operation `true[o]` used in the test at line 4 consists of a series of JavaScript primitive operations as follows:

(1) It applies ToObject to `true` to dynamically converts the value to an object. The semantics of ToObject creates an instance of `Boolean` intrinsic object by calling `new Boolean(true)`, and an internal field [[Prototype]][6] of the created object is set to the reference of `Boolean.prototype`.
(2) It applies the internal operation ToString to the field name designator `o` to dynamically convert the value to a string. The semantics of ToString *implicitly accesses* the field `toString` of the object `o` and calls it. As we defined at line 2, the function returns a string `"x"`.
(3) The field lookup operation searches a field named `"x"` from the instance of `Boolean` intrinsic object. Since the instance does not have a field named `"x"`, it *implicitly accesses* the field `"x"` of the object `Boolean.prototype`, which is the prototype chain object of the instance. Since the field value of the object `Boolean.prototype` is `10` as we defined at line 1, the result of the field lookup operation is evaluated to `10` and the equality test succeeds.

The evaluation of a field lookup operation in JavaScript involves two implicit field accesses, and by one of these field accesses reads a field value of the intrinsic object `Boolean.prototype`.

Our goal is to run programs *as if they were running in isolated realms*. For that we will actually run them in isolated *virtual realms* that rely on two ingredients: *i)* newly created controlled global and intrinsic objects, and, *ii)* a rerouting of the implicit field accesses from the original intrinsic objects to the controlled intrinsic objects.

In order to reroute field accesses, we use the two steps code transformation. First, we transform programs into equivalent ones but were implicit operations have be replaced with equivalent explicit ones. For instance, in the above example, the implicit field accesses are involved in the dynamic type conversion. For example, the code `true[o] === 10` is converted as follows:

```
1 // ToObject(true)
2 var r = new Boolean(true);
3 // ToString(o)
4 var s = o.toString();
5 // actual field lookup
6 function Load(o,s) {
7   if (o.hasOwnProperty(s)) return o[s];
8   o = Object.getPrototypeOf(o);
9   return Load(o,s);
10 }
11 Load(r,s) === 10;
```

Second, we rewrite the code `Load` to reroute the field accesses as follows:

[4]We intentionally do not provide `eval` and `with` constructs.
[5]https://gitlab.inria.fr/securejs/sjs-compiler

[6]In ECMAScript specification, internal field and method names are presented with a double square brackets [[Name]]. These methods and field names are not allowed to be called or accessed directly from JavaScript programs but callable or accessible from other internal methods or operations.

```
1 function SJSLoad(o,s) {
2   if (o.hasOwnProperty(s)) return o[s];
3   o = Object.getPrototypeOf(o);
4   if (o === Boolean.prototype)
5     o = isolated.Boolean.prototype;
6   return SJSLoad(o,s);
7 }
8 SJSLoad(r,s) === 10;
```

This code uses the `SJSLoad` instead of `Load` for the field lookup operation. The crucial point is line 4. When a field access reaches an intrinsic object `Boolean.prototype`, it reroutes the field access to the controlled intrinsic object `isolated.Boolean.prototype` (line 5). Thus, the rewritten code runs as if the code is running with the controlled intrinsic object.

In the rest of the paper, we propose a core JavaScript language, JS$^{\text{explicit}}$, that does not involve implicit field accesses, and we present a translation from JavaScript to JS$^{\text{explicit}}$ (Section 3). Then, we implement a secure runtime environment that consists of an isolated virtual realm and secure primitive operations which reroute field accesses to the virtual realm (Section 4).

3 JS$^{\text{explicit}}$, A CORE JAVASCRIPT LANGUAGE

In this section, we define JS$^{\text{explicit}}$, a core language for JavaScript, whose semantics is simpler than that of JavaScript in that its primitive operations do not involve implicit field access.

3.1 Syntax

We use the following formal syntax for JS$^{\text{explicit}}$:

$$
\begin{array}{rcl}
e & ::= & x \mid x_g \mid v \in \mathbb{V}_{\text{prim}} \mid \texttt{this} \mid e \oplus e \mid \ominus e \\
& & \mid \diamond\texttt{hasOwnProperty}\ e_1\ e_2 \mid \diamond\texttt{getPrototypeOf}\ e \\
& & \mid \diamond\texttt{toString}\ e \mid \diamond\texttt{toNumber}\ e \mid \diamond\texttt{toBoolean}\ e \\
i & ::= & \texttt{var}\ x \mid x = e \mid x_g = e \mid x_1 = \diamond\texttt{alloc}(x_2) \\
& & \mid e_1[e_2] = e_3 \mid x = e_1[e_2] \\
& & \mid x = \lambda(\vec{x}_n).\, i \mid \texttt{return}\ e \\
& & \mid x = \diamond\texttt{apply}(x_1, x_2, [\vec{y}_n]) \\
& & \mid \texttt{while}\ x\ \texttt{do}\ i \mid \texttt{if}\ (x)\ i_1\ \texttt{else}\ i_2 \\
& & \mid \texttt{try}\ i_1\ \texttt{catch}(x)\ i_2\ \texttt{finally}\ i_3 \\
& & \mid \texttt{throw}\ e \mid \{i\} \mid i_1; i_2 \mid \epsilon \\
& & \mid \diamond\texttt{defineOwnProperty}(x, e_1, e_2) \\
& & \mid x = \diamond\texttt{getOwnPropertyDescriptor}(e_1, e_2)
\end{array}
$$

where $\oplus$ and $\ominus$ are the binary and unary operators of the JavaScript language, respectively.

Expressions. An expression e is either a local variable access x, a global variable access x_g, a constant value $v \in \mathbb{V}_{\text{prim}}$, this value `this`, binary operators, or unary operators.

Non-standard Expressions. The operators whose name starts with the $\diamond$ symbol are non-standard operators. Their semantics follow those of the corresponding JavaScript built-in functions, but they do not involve implicit type conversion. The field test operator $\diamond$`hasOwnProperty` e_1 e_2 returns `true` if an object designated by e_1 has a field named e_2, and otherwise, returns `false`. This operator requires an object value for e_1 and a string value for e_2. The operator $\diamond$`getPrototypeOf` e returns a prototype chain object of the given object e. The operator requires an object value for e. The primitive type conversion operators $\diamond$`toString` e, $\diamond$`toNumber` e, and $\diamond$`toBoolean` e take a primitive value e as its argument and return a string, a number, and a boolean value, respectively.

Instructions. JS$^{\text{explicit}}$ uses non-standard semantics for the field lookup and update operations. The field update operation $e_1[e_2] = e_3$ updates a field designated by the value of e_2 of an object e_1 to the value of e_3. Note that the update operation does not access a prototype chain of the given object and does not involve a dynamic type conversion. Thus, e_1 must be evaluated to an object, and e_2 must be evaluated to a string value. The field lookup operation $x = e_1[e_2]$ reads a field designated by e_2 of an object e_1 and assigns the result to the local variable x. Like the field update operation, the field lookup operation does not access a prototype chain object and does not involve a dynamic type conversion. Thus, e_1 must be evaluated to an object, and e_2 must be evaluated to a string value.

The semantics of the rest of the instructions follow the standard semantics but these instructions do not involve dynamic type conversions. The variable declaration `var` x declares a variable in a local scope. The assignment instructions $x = e$ and $x_g = e$ assign the value of expression e to the local variable x and the global variable x_g, respectively. When the instruction assigns a value to a global variable, it updates to the new value if its writable property is `true`; otherwise, it ignores the update operation. The object allocation $x_1 = \diamond\texttt{alloc}(x_2)$ allocates an object whose a prototype chain designates x_2, and it assigns the newly allocated location to x_1. The function declaration $x = \lambda(\vec{x}_n).\, i$ allocates a new function object with its argument names $\vec{x}_n$ and the function body instruction i. We assume that all the program flows in the function body i end with a return instruction `return` e. The return instruction `return` e returns the value of the expression e to its caller site. The function call $x = \diamond\texttt{apply}(x_1, x_2, [\vec{y}_n])$ calls a function x_1 with a `this` value x_2 and its arguments $\vec{y}_n$, and it assigns the return value of the function to the variable x. The loop `while` x `do` i iterates its body i while the value of its condition x is true. The branch instruction `if` (x) i_1 `else` i_2 executes the instruction i_1 or i_2 depending on the value of variable x. The instruction `try` i_1 `catch`(x) i_2 `finally` i_3 executes the instruction i_1, and if there is an exception from the instruction i_1, it catches the exception value, bind it to a local variable x, and run the instruction i_2. Then, it executes the instruction i_3. The throw instruction `throw` e throws an exception with an exception value e. Once an exception is thrown, it ignores instructions until it reaches to the catch clause. The sequence $i_1; i_2$ executes the instructions i_1 and i_2 in order. The no-op instruction ϵ does nothing.

Non-standard Instructions. JS$^{\text{explicit}}$ uses non-standard instructions: $\diamond$`defineOwnProperty` and $\diamond$`getOwnPropertyDescriptor`. In JavaScript, they are supported by built-in functions `Object.defineProperty` and `Object.getOwnPropertyDescriptor`. The immediate field update operation with attributes $\diamond\texttt{defineOwnProperty}(x, e_1, e_2)$ assigns the value of the expression e_2 to an object x of a field e_1. The value of expression e_2 is either a data descriptor object that has fields *value*, *enumerable*, *configurable*, and *writable* attributes or a accessor descriptor object that has fields *get*, *set*, *enumerable*, and *configurable*. The immediate field lookup $x = \diamond\texttt{getOwnPropertyDescriptor}(e_1, e_2)$ reads a field e_2 of an object e_1 and assigns the field value record as an object to the variable x.

For the sake of simplicity, we use the following notations in the rest of the section: $x = f(y_1, \ldots, y_n)$ denotes $x = \diamond\texttt{apply}(f,\ \texttt{null},\ \vec{y}_n)$, and we use a function `isObject` defined as follows:

```
1 isObject = λ[x]. {
```

```
  return (typeof x === "object" && x !== null) ||
         typeof x === "function";
}
```

We omit else branches of `if` statements when empty.

3.2 Semantics

We define the semantics of a program by the means of a big-step semantics. The semantics of the JS$^{\text{explicit}}$ follows the semantics of JavaScript language, but they do not involve dynamic type conversion and implicit field access. The configurations have the form $\mu, l_s, i, t \Downarrow \mu', t'$, where $\mu, \mu' \in \mathbb{M}$ are memories, $l_s \in \mathbb{V}_{\text{loc}}$ is a current scope object location, i is an instruction, and t is either a normal state or an exception state with a value exc(v) where $v \in \mathbb{V}$. The state tag normal denote that the program state is in a normal state, and the state tag exc(v) denote that the program state is in an exception state with an exception value v. A memory $\mathbb{M} = \mathbb{V}_{\text{loc}} \rightarrow \mathbb{O}$ is a map from locations to objects and an object $o \in \mathbb{O} = \mathbb{V}_{\text{fld}} \rightarrow \mathbb{R}$ is a map from field names to field values where $\mathbb{V}_{\text{fld}} = \mathbb{V}_{\text{str}} \cup \mathbb{V}_{\text{i}}$ and $\mathbb{V}_{\text{str}}$ are strings and $\mathbb{V}_{\text{i}}$ are special values used to name internal JavaScript fields such as *@env* and *@proto*, and $\mathbb{R} = \mathbb{V}_{\text{str}} \rightarrow \mathbb{V}$ is a map from record names to values and we write $\{name_1 : v_1, name_2 : v_2\}$ to denote a record that has two elements of names $name_1$ and $name_2$ respectively with its values v_1 and v_2.

The evaluation of an expression e has the form $\mu, l_s, e \Downarrow_{\text{E}} v$, where $\mu \in \mathbb{M}$ is a program memory, v is a value of the expression in the memory.

The auxiliary function Put(l_s,x,v) either updates the current scope object with $x = v$ or walks the scope chain designated by the outer scope *@env* to update the value of a local variable x as defined by:

$$\frac{\mu' = \mu[l_s \mapsto \mu(l_s)[x \mapsto v]]}{\mu,\ \text{Put}(l_s,\ x,\ v) \Downarrow_{\text{A}} \mu'}\ x \in \text{dom}(\mu(l_s))$$

$$\frac{\mu,\ \text{Put}(\mu(l_s)(@env).1,\ x,\ v) \Downarrow_{\text{A}} \mu'}{\mu,\ \text{Put}(l_s,\ x,\ v) \Downarrow_{\text{A}} \mu'}\ x \notin \text{dom}(\mu(l_s))$$

Figure 1 presents an excerpt of the formal semantics containing only the essential rules. It shows the semantics of object allocation, field lookup, and update operations. Unlike the standard JavaScript semantics, the field lookup and update operations only accept that a receiver is an object and a field name designator evaluates to a string value, and the operations do not search a field in a prototype-chain of the given receiver object.

- **Alloc rule.** The instruction x_1 `= ◇alloc(`x_2`)` evaluates the argument x_2 to a value v, allocates a new object in a new location l_n with a prototype chain value v, and assigns the allocated object reference to the local variable x_1. The instruction requires the prototype value x_2 to either be a location value or a `null` value.
- **Lookup rule.** The instruction x_1 `=` x_2`[`e`]` first evaluates the receiver object x_2 and field name designator e respectively to a location l and a string value k. The semantics do not search a prototype chain, but merely read a field of the receiver object $\mu(l)(k)$. In this rule, the field must be a record, which has four fields *value*, *enumerable*, *configurable*, and *writable*, and it assigns the value of the field *value* to the left-hand side variable x_1 by calling the auxiliary function Put.
- **Update rule.** The instruction x`[`e_1`]` `=` e_2 also evaluates the receiver object x to a location l, the field name e_1 to a string value k. It evaluates the expression e_2 to a value v, and it assigns the value v as a record with enumerable, configurable, and writable fields to the field k of the receiver object designated by the location l. This rule is applied when there was no field named k in the receiver object, and it creates a new field.

The semantics of non-standard instructions and operators follows the semantics of the corresponding JavaScript built-in functions, but they do not apply dynamic type conversion.
Figure 2 shows three additional semantics rules:

- **hasOwnProp1 and hasOwnProp2 rules**. The binary operation `◇hasOwnProperty` tests whether the object e_1 has a field designated by the string name e_2 or not.
- **getPrototypeOf rule**. The unary operation `◇getPrototypeOf` e returns the internal prototype chain value of the given object.

The unary operations `◇toString`, `◇toNumber`, and `◇toBoolean` take a primitive value as an argument and returns their corresponding typed value. The conversion algorithm is defined in the ECMAScript specification[7] [4], and these conversion algorithm do not involve implicit field access.

3.3 JavaScript Primitive Operations in JS$^{\text{explicit}}$

In this section, we present JavaScript primitive operations written in JS$^{\text{explicit}}$. Each primitive operation is devised by a corresponding internal method or operation defined in ECMAScript specification. We write [[A]] and B to denote an internal field or method named A and an operation named B, respectively.

Figure 3a shows the [[Get]] internal method of ECMAScript specification that defines the essential semantics of a field lookup operation in JavaScript. For a given code is `o[s]`, `o` is passed as *base* and `s` is passed as P in the specification. Figure 3b shows the code of `Get` function which mimics the semantics of [[Get]] internal method in JS$^{\text{explicit}}$. Each step of the auxiliary function `Get` is designed with the specification and written in JS$^{\text{explicit}}$.

First, the [[Get]] internal method of the specification converts the receiver value *base* to an object by calling the ToObject operation (step 1). In the `Get` function, it calls the function `ToObject` which mimics the semantics of the operation ToObject in the specification. This operation returns the input value if a type of the input value is already an object. Otherwise, it allocates a typed object if a type of the input value is one of string, number, and boolean types. This semantics is defined by the auxiliary function `ToString` written in JS$^{\text{explicit}}$. Then, it calls [[GetProperty]] internal method to search the field named P from the object *base* and its prototype chain (step 2). The result is either `undefined` value when it fails to find a field or a descriptor object. If the result is `undefined`, it returns `undefined` value (step 3). The descriptor object has `value` field if the field is an ordinary value field. Otherwise, the descriptor object has to have the `get` field and it is the getter value. Thus, it first checks whether the descriptor object has a field `"value"`, and it returns the value of the field `"value"` when the descriptor object has the field (step 4).

[7] ToBoolean: https://www.ecma-international.org/ecma-262/5.1/#sec-9.2,
ToNumber: https://www.ecma-international.org/ecma-262/5.1/#sec-9.3,
ToString: https://www.ecma-international.org/ecma-262/5.1/#sec-9.8

$$\textsc{Alloc}\ \frac{\mu, l_s, x_2 \Downarrow_{\mathrm{E}} v \qquad l_n \text{ fresh in } \mathbb{V}_{\mathrm{loc}} \qquad \mu' = \mu[l_n \mapsto \{@proto \mapsto \{value : v\}\}] \qquad \mu', \mathrm{Put}(l_s, x_1, l_n) \Downarrow_{\mathrm{A}} \mu''}{\mu, l_s, x_1 \ = \ \diamond\texttt{alloc}(x_2), \text{normal} \Downarrow \mu'', \text{normal}}$$

$$\textsc{Lookup}\ \frac{\mu, l_s, x_2 \Downarrow_{\mathrm{E}} l \qquad \mu, l_s, e \Downarrow_{\mathrm{E}} k \qquad v = \mu(l)(k).value \qquad \mu, \mathrm{Put}(l_s, x_1, v) \Downarrow_{\mathrm{A}} \mu'}{\mu, l_s, x_1 \ = \ x_2[e], \text{normal} \Downarrow \mu', \text{normal}}$$

$$\textsc{Update}\ \frac{\mu, l_s, x \Downarrow_{\mathrm{E}} l \qquad \mu, l_s, e_1 \Downarrow_{\mathrm{E}} k \qquad \mu, l_s, e_2 \Downarrow_{\mathrm{E}} v \qquad \mu' = \mu[l \mapsto \mu(l)[k \mapsto \{value : v, enumerable : \texttt{true}, configurable : \texttt{true}, writable : \texttt{true}\}]]}{\mu, l_s, x[e_1] \ = \ e_2, \text{normal} \Downarrow \mu', \text{normal}} \ \textit{where } k \notin \textit{dom}(\mu(l))$$

Figure 1: An excerpt from semantics rules for instructions

$$\textsc{hasOwnProp1}\ \frac{\mu, l_s, e_1 \Downarrow_{\mathrm{E}} l \qquad \mu, l_s, e_2 \Downarrow_{\mathrm{E}} k \qquad k \in \mathrm{dom}(\mu(l))}{\mu, l_s, \diamond\texttt{hasOwnProperty}\ e_1\ e_2 \Downarrow_{\mathrm{E}} \texttt{true}}$$

$$\textsc{hasOwnProp2}\ \frac{\mu, l_s, e_1 \Downarrow_{\mathrm{E}} l \qquad \mu, l_s, e_2 \Downarrow_{\mathrm{E}} k \qquad k \notin \mathrm{dom}(\mu(l))}{\mu, l_s, \diamond\texttt{hasOwnProperty}\ e_1\ e_2 \Downarrow_{\mathrm{E}} \texttt{false}}$$

$$\textsc{getPrototypeOf}\ \frac{\mu, l_s, e \Downarrow_{\mathrm{E}} l \qquad v = \mu(l)(@proto).value}{\mu, l_s, \diamond\texttt{getPrototypeOf}\ e \Downarrow_{\mathrm{E}} v}$$

Figure 2: An excerpt from semantics rules for expressions

When the [[Get]] internal method is called using *base* as its `this` value and with property name *P*, the following steps are taken:

(1) Let *O* be ToObject(*base*).
(2) Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
(3) If *desc* is `undefined`, return `undefined`.
(4) If IsDataDescriptor(*desc*) is `true`, return *desc*.[[Value]].
(5) Otherwise, IsAccessorDescriptor(*desc*) must be `true` so, let *getter* be *desc*.[[Get]] (see 8.10).
(6) If *getter* is `undefined`, return `undefined`.
(7) Return the result calling the [[Call]] internal method of *getter* providing *base* as the `this` value and providing no arguments.

(a) [[Get]] internal method

```
1 Get = λ(O,P). {
2   var result, getter;
3   O = ToObject(O); /* (1) */
4   var desc = GetProperty(O,P); /* (2) */
5   if (desc === undefined) return undefined; /* (3) */
6   if (◇hasOwnProperty desc "value") { /* (4) */
7     result = desc["value"];
8     return result
9   } else {
10    getter = desc["get"]; /* (5) */
11    if (getter === undefined) return undefined; /* (6) */
12    result = ◇apply(getter, O, []); /* (7) */
13    return result
14  }
15 }
```

(b) The function `Get` written in JS$^{\text{explicit}}$

Figure 3: The specification of [[Get]] internal method and its meaning written in JS$^{\text{explicit}}$

Otherwise, it loads the field `"get"` (step 5). If the getter function is `undefined`, it just returns `undefined` value (step 6). Otherwise, the getter function is a function value, and it calls the getter function with *base* as its `this` value and returns the result (step 7).

Figures 4a and 4b show the [[GetProperty]] internal method specification and its meaning written in JS$^{\text{explicit}}$. The [[GetProperty]] method defines the field search in a prototype chain. It first tries to access the field *P* from the given object *O* (step 1). If the object *O* has a field named *P*, it just returns the field record (step 2). Otherwise, it loads the value of a prototype chain object of the object *O* (step 3). If the given object does not have a prototype chain object, it fails to find a field and returns `undefined` value (step 4). Otherwise, it recursively searches the field named *P* from the prototype chain object of the object *O* (step 5). The auxiliary function `GetProperty` implements the semantics in JS$^{\text{explicit}}$.

Figures 5a and 5b show the [[DefaultValue]] internal method specification with hint String and its meaning written in JS$^{\text{explicit}}$. The [[DefaultValue]] method defines the dynamic type conversion when a given value is an object. In a field lookup operation `o[s]`, when the given field name designator `s` is an object, it converts the field name value `s` to a string typed value by calling the [[DefaultValue]] method. In the [[DefaultValue]] method, it first calls [[Get]] internal method with the argument `toString` and we define its corresponding auxiliary function in Figure 3a (step 1). If the result is a callable, that is, if the value is a function (step 2), it calls the result function with `O` as the this value (step 2a). If the result of `toString` is a primitive value, it returns the result (step 2b). Otherwise, it calls [[Get]] internal method with an argument `valueOf` (step 3). If the result is a callable (step 4), it calls the result function with `O` as the this value (step 4a). If the result of `valueOf` is a primitive value, it returns the result (step 4b). If both result values of `toString` and `valueOf` are not a primitive value, it throws a `TypeError` exception (step 5). In the auxiliary function written JS$^{\text{explicit}}$, it creates an instance of `TypeError`, which designates a built-in function of `TypeError` object.

Figure 6 presents the semantics of ToString and ToNumber operations defined in ECMAScript specification. If the value is not an object, it converts the primitive value respectively to a string value and a number value by `◇toString` and `◇toNumber` operations. Otherwise, it calls `DefaultValue` with a hint "String" or "Number", which mimics the semantics of [[DefaultValue]]. Then, it again calls the function itself `ToString` or `ToNumber` with the result of `DefaultValue`. Since the auxiliary function `DefaultValue` always returns a primitive value, this recursion does not create a loop, and the auxiliary

When the [[GetProperty]] internal method of *O* is called with property name *P*, the following steps are taken:

(1) Let *prop* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
(2) If *prop* is not undefined, return *prop*.
(3) Let *proto* be the value of the [[Prototype]] internal property of *O*.
(4) If *proto* is null, return undefined.
(5) Return the result of calling the [[GetProperty]] internal method of *proto* with argument *P*.

(a) [[GetProperty]] internal method

```
GetProperty = λ(O,P). {
  var prop, proto, result;
  prop = ⋄getOwnPropertyDescriptor(O,P); /* (1) */
  if (prop !== undefined) { return prop } /* (2) */
  else {
    proto = ⋄getPrototypeOf(O); /* (3) */
    if (proto === null) { return undefined } /* (4) */
    else {
      result = GetProperty(proto, P); /* (5) */
      return result
    }
  }
}
```

(b) The function `GetProperty` written in JS^{explicit}

Figure 4: The specification of [[GetProperty]] internal method and its meaning written in JS^{explicit}

When the [[DefaultValue]] internal method of *O* is called with hint String, the following steps are taken:

(1) Let *toString* be the result of calling the [[Get]] internal method of object *O* with argument "toString".
(2) If IsCallable(*toString*) is true then,
  (a) Let *str* be the result of calling the [[Call]] internal method of *toString*, with *O* as the this value and an empty argument list.
  (b) If *str* is a primitive value, return *str*.
(3) Let *valueOf* be the result of calling the [[Get]] internal method of object *O* with argument "valueOf".
(4) If IsCallable(*valueOf*) is true then,
  (a) Let *val* be the result of calling the [[Call]] internal method of valueOf, with *O* as the this value and an empty argument list.
  (b) If *val* is a primitive value, return *val*.
(5) Throw a TypeError exception.

(a) [[DefaultValue]] internal method

```
DefaultValue = λ(O,hint). {
  var str, val, toString, valueOf;
if (hint === "String") {
    toString = Get(O, "toString"); /* (1) */
    if (typeof toString === "function") { /* (2) */
      str = ⋄apply(toString, O, []); /* (2a) */
      if (!isObject(str)) return str else { } /* (2b) */
    } else { }
    valueOf = Get(O, "valueOf"); /* (3) */
    if (typeof valueOf === "function") { /* (4) */
      val = ⋄apply(valueOf, O, []); /* (4a) */
      if (!isObject(val)) return val else { } /* (4b) */
    } else { }
    throw TypeError();   /* (5) */
  } else { /* hint === "Number" */
    /* it calls "valueOf" and "toString" in order. */
  }
}
```

(b) The function `DefaultValue` written in JS^{explicit}

Figure 5: The specification of [[DefaultValue]] internal method and its meaning written in JS^{explicit}

```
ToString = λ(x). {
  if (!isObject(x)) { return ⋄toString(x) }
  else {
    x = DefaultValue(x, "String");
    x = ToString(x);
    return x
  }
}
```

(a) The auxiliary function `ToString`

```
ToNumber = λ(x). {
  if (!isObject(x)) { return ⋄toNumber(x) }
  else {
    x = DefaultValue(x, "Number");
    x = ToNumber(x);
    return x
  }
}
```

(b) The auxiliary function `ToNumber`

Figure 6: The ToString and ToNumber in JS^{explicit}

functions `ToString` and `ToNumber` return a string and number value, respectively.

Support for ECMAScript built-in functions. The ECMAScript built-in functions also rely on the primitive operations defined above, and the semantics involves implicit field access. For instance, let us consider the `slice` method of the String API. According to the specification [4], when this method is invoked, the `this` argument must be converted into a string and the arguments into integers. These conversions have to be protected too. For that, the original `slice` function (`String_prototype_slice`) is protected by a JS^{explicit} wrapper defined as:

```
wrap = λ(old_slice). {
  var new_slice = λ[start,end].{
    var s;
    CheckObjectCoercible(this);
    s = ToString(this);
    start = ToInteger(start);
    if (end !== undefined)
      end = ToInteger(end);
    return ⋄apply(old_slice, s, [start,end]);
  }
};
/* wrapped String.prototype.slice */
slice = wrap(String_prototype_slice);
```

All JavaScript functions involving conversion and field accesses are wrapped similarly.

3.4 Translation to JS^{explicit}

We present translation function compile⟦*P*⟧ that takes JavaScript code *P* and translates it into JS^{explicit}.

First, we consider explicit references to global variables, e.g., `Object`, `Function`, or `String`. These explicit references are replaced

with accesses to the global object, and later on, we will present a revised operation to reroute these accesses to a virtual global object. When an identifier expression does not have a corresponding variable declaration binding, the expression is considered as a global variable access. compile⟦x_g⟧ transforms a global variable access x_g into an auxiliary function call `Loadvar("x_g")` defined as follows:

```
1 Loadvar = λ(id).{
2   Get(global, id);
3 }
```

The variable `global` designates the global object. Note that the semantics of a global variable read searches the global object and its prototype chain.

We consider a translation of a field lookup operation compile⟦x = e1[e2]⟧ where x is a local variable. The translated code to $JS^{explicit}$ is as follows:

```
1 fn = λ(o,s).{
2   if (o === null || o === undefined)
3     throw TypeError();
4   s = ToString(s);
5   x = Get(o, s);
6 };
7 fn(compile⟦e1⟧, compile⟦e2⟧)
```

A field lookup operation first ensures that the given receiver o is a *coercible* value. If the receiver object is `null` or `undefined`, it throws a `TypeError` exception. Otherwise, the value is either an object value or able to be converted to an object value in the auxiliary function `Get`. Then, it converts the field name designator s to a string value by calling an operation ToString, and we use corresponding auxiliary function `ToString` presented in 6a. At last, it uses an auxiliary function for a field lookup operation `Get` and assigns the result to the local variable x.

Rewriting explicit field accesses is not enough to prevent implicit field accesses as JavaScript accesses fields *implicitly* in many situations. For instance, let us consider the binary operation `e1-e2`. According to the JavaScript semantics, `e1` and `e2` have to be converted to a number before applying the subtraction. How to make the conversion depends on the type of `e1` and `e2`. If it is an object, its `valueOf` field is used, That is, if one of `e1` and `e2` is an object, the subtraction operation involves implicit access to the `valueOf` field. This access has to be intercepted and replaced as explicit field accesses. For that, a translation of compile⟦e1-e2⟧ to $JS^{explicit}$ is defined as follows:

```
1 fn = λ(lhs,rhs).{
2   lhs = ToNumber(lhs);
3   rhs = ToNumber(rhs);
4   return lhs - rhs;
5 };
6 fn(compile⟦e1⟧, compile⟦e2⟧)
```

The auxiliary function `ToNumber` presented in Figure 6b guarantees that the result is a number value. Thus, the actual subtract operation `lhs - rhs` at line 4 of the translated code does not involve implicit field access.

In JavaScript, a function invocation is considered as a method invocation if and only if the syntactic form of the function is an object field access. That is, "`obj.met(arg)`" is a method invocation and the pseudo argument `this` will be bound to `obj` in the body of `obj.met`. On the other hand, in "`var f=obj.met; f(arg)`" is not a method invocation but as a regular function call and `this` will not be bound to `obj`. This difference between function invocation and method invocation demands the $JS^{explicit}$ compilation to specially treat method invocation. We consider a translation of a method call expression compile⟦x = e1[e2](e3)⟧ where x is a local variable as follows:

```
1 fn = λ(o,s).{
2   var f;
3   f = compile⟦o[s]⟧;
4   return ◇apply(f, o, compile⟦e3⟧);
5 };
6 x = fn(compile⟦e1⟧, compile⟦e2⟧);
```

First, the syntactic part `e1[e2]` is transformed as regular field lookup operation. Second, when the function is found in the object, the function is invoked in such a way that `e1` (the first argument o) is bound to `this` and `e3` as the argument.

4 IMPLEMENTATION: SecureJS

In this section, we present the SecureJS implementation. SecureJS compiled programs execute as if they were in isolated realms, which do not share any references with other realms. Thus, if a compiled program does not export value to others, executions of the compiled program preserve total memory isolation, and it guarantees that running a compiled program with other uncompiled programs does not interfere with each other.

This isolation relies on a secure runtime environment, which consists of an isolated virtual global object and intrinsic objects, and secure primitive operations, which reroute field accesses for the global and intrinsic objects to corresponding virtual ones. This section details the implementation the secure primitive operations and the runtime environment.

4.1 Secure Primitive Operations

The SecureJS compiler rewrites JavaScript programs in such a way that no expression can ever access, explicitly or implicitly, objects created by uncompiled code. For that, the compiled code replaces normal object field access and normal prototype chain lookups with custom operations that enforce memory isolation. Let us detail these operations.

First of all, explicit references to global variables must be replaced with accesses to secure objects that proxy these global plain objects. For that, the compiler transforms a global variable access x into a library function call `SJS.Loadvar("x")` defined as:

```
1 SJS.Loadvar = function SJSloadvar(id) {
2   return Get(SJS.global, id);
3 }
```

The object `SJS.global` is a secure version of the global object. It ensures that all the interactions with the global object will not be exposed to the uncompiled code.

Second, object field accesses have to be intercepted in order to prevent the normal prototype chain inspection to access objects of the uncompiled environment. For that, we provide a secure version of `GetProperty` auxiliary function defined as:

```
1 SJS.GetProperty = function SJSGetProperty(O, P) {
2   var prop;
3   prop = SJS.getOwnPropertyDescriptor(O,P); /* (1) */
4   if (prop !== undefined) return prop; /* (2) */
```

```
5   else {
6     O = SJS.getPrototypeOf(O); /* (3) */
7     if (O === null) return undefined; /* (4) */
8     if (SJS.securePrototypeOf(O) !== undefined)
9       O = SJS.securePrototypeOf(O)
10    return SJSGetProperty(O,P); /* (5) */
11  }
12 }
```

The crucial point is line 8. It ensures that the field lookup will never escape the memory controlled by the compiled environment. If the search for the field hits the global object that belongs to the uncompiled code, it reroutes the search to another object (line 9) under the control of SecureJS.

For compatibility with current IoT platforms such as JerryScript [5], SecureJS compiles and targets ECMAScript 5 but it also supports a restricted version of the ECMAScript 6 module export form. The `export` instruction to establish communication channels between compiled and uncompiled code. The compiler transforms a declaration "`export id`" into:

```
id = SJS.Loadvar("id")
```

where the left-hand side `id` designates the global variable `id` outside the realm.

The last role of the compiler is to create a secure execution context for the compiled code. For that, the compiled code will be embedded into an anonymous function that will start creating the secure environment and that will then execute client code. That is, all compiled programs have the following shape:

```
(function() {
   var SJS = SJSbootstrap();
    ... client compiled code ...
})();
```

SecureJS compiled code guarantees that any uncompiled expression cannot ever access objects created by compiled code. The variable `SJS` is hidden in the local scope of the compiled code, and the function `SJSbootstrap` is immutable and returns a new isolated realm whenever it is called.

4.2 Secure Runtime Environment

SecureJS compiled code initializes a secure environment by invoking the `SJSbootstrap` (see Figure 7, line 42) function at the very beginning of the execution. This function is defined in a library `sjs.js` that captures native built-in objects and provides secure interfaces for compiled programs. In this section, we present the implementation of that library.

The `sjs.js` library executes before any other computation. It captures the standard JavaScript functions (line 2-5) and stores them in a function local variables so that they cannot be tampered from outside the library. It creates an immutable field `@call` to `SJScall` with the built-in value of `Function.prototype.call` (line 6) so that the semantics of the call expression `SJScall['@call']` also cannot be tampered, and the `SJScall['@call'](f,obj,args)` implements `◇apply(f,obj,args)` in $JS^{explicit}$. The library defines the function `SJSbootstrap` (line 9) that is bound to the global JavaScript variable `SJSbootstrap` (line 42). The function definition requires two steps, as this is the only way ECMAScript 5 proposes to define read-only variables.

```
1 (function() {
2   var SJScreate = Object.create;
3   var SJScall = Function.prototype.call;
4   var SJSObjProto = Object.prototype;
5   var SJSFunProto = Function.prototype;  ...
6   Object.defineProperty(SJScall, '@call',
7      { value: SJScall, configurable: false,
8        writable: false, enumerable: false });
9   function SJSbootstrap() {
10    function securePrototypeOf(obj) {
11      if(obj === SJSObjProto)
12        return SecureObjP;
13      if(obj === SJSFunProto)
14        return SecureFunP;
15      ...
16      return false;
17    }
18    var global = SJScreate(null);
19    var SJS = SJScreate(null);
20    SJS.global = global;
21    function CheckObjectCoercible(o) {...};
22    function Get(x) {...};
23    function DefaultValue_S(x) {...};
24    function ToString(o) {...};
25    function ToNumber(o) {...};
26    function ToInteger(o) {...};
27    SJS.Loadvar = function (id) {...};
28    SJS.GetProperty = function (O, P) {...};
29    global.Object = function (v) {...};
30    ...
31    global.String.prototype.slice =
32      function(v) {...};
33    var SecureObjP =
34      global.Object.prototype =
35        SJScreate(null);
36    var SecureFunP =
37      global.Function.prototype =
38        SJScreate(SecureObjP);
39    ...
40    return SJS;
41  };
42  Object.defineProperty(this,
43    "SJSbootstrap", {
44     value:SJSbootstrap, configurable:false,
45     writable:false, enumerable:false });
46 })();
```

Figure 7: An excerpt of `sjs.js`.

The function `SJSbootstrap` creates a regular JavaScript object (line 18) that will act as a replacement of the standard JavaScript global object inside the secure code.

5 EVALUATION

This section seeks for an experimental validation that the SecureJS compiler preserves the semantics of the original program and the compiled code complies with memory isolation. First, we test that SecureJS compilation preserves the semantics of the original program in practical JavaScript engines V8 [6], Hop [12], and JerryScript [5] using the Test262 test suite. Second, we test that a SecureJS compiled program complies with memory isolation by testing that execution of the compiled program does not cause implicit field accesses. For this, we used the Test262 test suite in Hop and JerryScript JavaScript engines.

5.1 Experimentation Settings

We implemented SecureJS on top of the front-end of the open-source JavaScript engine Hop compiler [12]. SecureJS takes a JavaScript program and generates an ECMAScript 5th specification compatible JavaScript program. Executions are done with state-of-the-art JavaScript engines V8, Hop, and JerryScript.

We use the Test262 test suite and use the `es5-tests` branch from the github repository. Since the test programs of the Test262 test suites check the program states and flows by means of the `assert` function, the comparison of the resulting state confirms the equivalent execution of the programs. The Test262 programs also test the semantics of all the ECMAScript built-in functions and the semantics of the `strict mode` functions. We consider 10490 numbers of the test programs as our target test programs of the Test262 test suite and exclude 1082 numbers of the test programs from the target programs. In particular, we excluded tests using the `eval` function, the `with` statement. SecureJS does not support `eval` and `with`. Additionally, we excluded tests checking syntactic errors. The latest version of ECMAScript specifies less strict syntactic rules than the syntactic rules specified in ECMAScript 5th. Since the front-end parser of SecureJS supports the latest ECMAScript specification for compatibility, it parses the program that follows the latest ECMAScript specification and translates it into an ECMAScript 5th compatible program when possible.

5.2 Testing Semantic preservation

The goal of this experiment is to show that SecureJS compiler preserves the semantics of an original program. For that, we compiled target test programs of the Test262 suites and ran them on V8, Hop, and JerryScript JavaScript engines, and we checked whether the resulting states of both the program and the corresponding compiled program are equivalent.

Among 10490 numbers of the test programs, V8 passed 10386 successfully. 104 failed because of the semantics differences between the source language test 262 assumed and the latest JavaScript evolutions V8 supports. SecureJS preserves this result. That is, the 10386 successful tests remain successful after SecureJS compilation, and the 104 failed tests remain failed. Hop passed 10414 numbers of the test programs and failed in 76 tests, and JerryScript passed 10474 numbers of the test programs and failed in 16 tests. SecureJS also preserves these results.

5.3 Testing Memory Isolation

The SecureJS compiled programs run as if they were running in an isolated realm. If the compiled programs do not intentionally export values for interaction with others, the compiled and other uncompiled programs cannot reference each other. In this section, we test that such SecureJS compiled programs that do not use `export` comply with this isolation property. For that, we empirically verify that compiled programs ever access objects outside the controlled realm.

For the experiment, we modified the Hop and JerryScript implementations to log all accesses to the [[Prototype]] field that is needed to walk prototype chains. We have used two radically different JavaScript implementations in order to reduce the probability of unnoticed [[Prototype]] accesses that would be removed by the JavaScript engine's optimizer, independently of SecureJS. In addition, we modified the JerryScript engine also to log all accesses to global and intrinsic objects, which are outside of the controlled realm.

We executed all the 10490 programs from the Test262 suite with the instrumented JavaScript engines. We confirmed that there is no field access attempt to global and intrinsic objects. We observed a single situation where [[Prototype]] is implicitly accessed by the programs compiled by SecureJS. During the initialization of a program, an empty object is created with an empty prototype chain by the expression `Object.create(null)`. Later on, this object is used to optimized SecureJS field accesses. Since this object has no prototype chain, the accesses are harmless but are logged by the instrumented engines.

6 RELATED WORK

A number of studies have been carried out so as to provide memory isolation in JavaScript.

Bhargavan *et al.* [2] proposes defensive JavaScript (DJS), which is a small subset of the JavaScript language satisfying static typing constraints. DJS achieves memory isolation by the use of a small subset of the JavaScript language that does not involve dynamic features. The main limitation of DJS is that legacy code needs to be manually rewritten in the DJS subset and be typable in order to achieve memory isolation. A program written in DJS does not rely on a realm and cannot use built-in functions defined in ECMAScript specification.

Swamy *et al.* [14] proposes JS*, which is a programming language supporting various dynamic features of JavaScript, while ensuring type safety in the presence of an untrusted JavaScript program. As it requires type-annotations for the trusted programs, legacy code needs to be manually rewritten in the JS*, and it supports only a small subset of JavaScript language in terms of the support of built-in functions. It implements a limited form of interfaces for inter-communication since the interface requires type-annotations and must implement the interface functions with the compiled interpreter programs.

Terrace *et al.* [15] implements a JavaScript interpreter written in JavaScript. It supports interactions that use primitive values, typed objects, and functions that has typed argument and return values, and it does not work on the JerryScript engine. For these reasons, it cannot be applied to IoT environments in particular.

Vasilakis *et al.* [16] proposes BreakApp, which provides isolation and interaction policies between programs. It relies on various levels of built-in isolation primitives, yet the isolation mechanisms strongly depends on Node.js. One of the isolation mechanisms BreakApp relies on is the isolated realm supported by Node.js.

Guha *et al.* [7] proposes a formal semantics of a core calculus, λ_{JS}, and provides a reduction step of JavaScript to λ_{JS}. This reduction step is similar to our translation to JS$^{\text{explicit}}$, but our translation to JS$^{\text{explicit}}$ focuses on rewriting primitive operations to their secure versions in order to simulate a JavaScript realm.

7 CONCLUSION

We presented SecureJS, a secure JavaScript-to-JavaScript compiler. It transforms programs into equivalent ones but that would execute

as if they were running inside an isolated realm. This compiler is portable and only uses standard features so it can be used by any JavaScript engine. In order to rewrite the semantics of dynamic features of JavaScript language, we presented $JS^{explicit}$, which does not involve implicit field accesses, and presented translation from JavaScript to $JS^{explicit}$. In the evaluation, we showed that the compilation preserves the semantics of the original programs that use full-features of ECMAScript 5.1 and achieves memory isolation if a given program does not export a value. In order to show that executions of the compiled programs preserve memory isolation, we use instrumented JavaScript engines, and to the best of our knowledge, no one has attempted to test such properties before by JavaScript engine instrumentation. SecureJS compiler provides an essential building block to use JavaScript security enforcement libraries that rely on isolated realms, in particular, it enables us to use such libraries in JerryScript JavaScript implementation, which does not support a built-in isolation mechanism.

REFERENCES

[1] Agoric. 2020. SES: Proposal for Secure EcmaScript. https://github.com/tc39/proposal-ses.

[2] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2013. Language-based Defenses Against Untrusted Browser Origins. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 653–670.

[3] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 87–100. https://doi.org/10.1145/2535838.2535876

[4] European Association for Standardizing Information and Communication Systems (ECMA). 2011. ECMA-262: ECMAScript Language Specification. Edition 5.1.

[5] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. 2015. Ultra Lightweight JavaScript Engine for Internet of Things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2015)*. ACM, New York, NY, USA, 19–20. https://doi.org/10.1145/2814189.2816270

[6] Google. 2018. V8 JavaScript Engine. http://developers.google.com/v8.

[7] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–150.

[8] Zhengqin Luo and Tamara Rezk. 2012. Mashic Compiler: Mashup Sandboxing Based on Inter-frame Communication. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium (CSF '12)*. IEEE Computer Society, Washington, DC, USA, 157–170. https://doi.org/10.1109/CSF.2012.22

[9] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. October 2007. Caja - safe active content in sanitized JavaScript. http://math.ucr.edu/~mike/caja-spec-2008-06-06.pdf.

[10] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 736–747.

[11] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety: Type-based Verification of JavaScript Sandboxing. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=2028067.2028079

[12] Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 180–192. https://doi.org/10.1145/2951913.2951916

[13] Patrik Simek. 2020. VM2 npm package. https://www.npmjs.com/package/vm2.

[14] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 425–437. https://doi.org/10.1145/2535838.2535889

[15] Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta. 2012. JavaScript in JavaScript (js.js): Sandboxing Third-Party Scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*. USENIX, Boston, MA, 95–100.

[16] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.